ISETL: A Programming Language for Learning Mathematics

Ed Dubinsky
Purdue University
In honor of Jack Schwartz on his 65th birthday.

1 Historical Introduction

It was back in 1969, at a functional analysis conference at the Indian Institute of Technology in Kanpur, India that I first heard about SETL from Jack Schwartz. It sounded fascinating, but I was not moved to look into it any further until a dozen years later when Jack was helping me get started with a training program for mathematicians who wanted to learn how to teach computer science (the IFRICS project). I wanted to find out more about SETL and he gave me a copy of an internal, unpublished report on the language. In that report I noted the following sentence.

> The [mathematical] knowledge assumed [for programming in SETL] is roughly equivalent to that which would be acquired in a freshman-level course in Discrete Mathematics.

What struck me about that sentence is the mismatch between the really large number of people who are willing and able to learn to program, even in a complicated language, and the much smaller group who are successful at assimilating even the most elementary concepts in discrete mathematics. Jack was referring to things like sets, sequences, functions (as algorithms or sets of ordered pairs), propositional and predicate logic. These are all topics that a successful SETL programmer knows. They are also concepts which give much trouble to students in mathematics courses.

The point is not trivial, because it was Jack's intention to base a programming language on fundemental concepts of mathematics. He felt that if one based the design of a language on finite sets and functions on them in a manner that was true to the mathematics involved, then the language would develop into something that could express the most complex mathematical relationships in a manner that added little or no difficulty to what was already contained in the mathematics. As we will see, he was not only right about that (as witness the accomplishments of SETL), but he also produced a unique and potentially effective pedagogical tool.

A language constructed on a mathematical foundation is sure to be a powerful problem solving tool — for those who can handle the mathematics. What, I wondered in 1981, about the others? If SETL programmers need to be reasonably facile in working with complex mathematics, does that not restrict access to this beautiful and expressive programming language?

Not as much as you would think. As I observed the world of computer science in the 1980s, it seemed to me that there were quite a few people who could use programming features based on design principles of which they were not necessarily aware. Suppose these programmers tried to pay attention to the design principles? Or were they learning things without explicitly focussing on these principles? I wondered if Pascal programmers got better at using the problem solving method of "divide and conquer", if Prolog programmers became good at logic, or if functional

programmers deepened their understanding of composition of functions. Whether or not this happened spontaneously, suppose instruction in mathematics made use of the mathematical foundations of a programming language? It occured to me that one could reverse Jack's point. One could set students the task of learning to program in SETL and then see if it was any easier for them to learn about the algebra of sets, existential and universal quantification, and the construction and manipulation of functions.

I decided to pursue this thought and the result has been very gratifying. I think it is fair to say that there is today, in the field of post-secondary mathematics, a collection of individuals, a number of projects and a body of published work involved with having students learn mathematics by writing programs in what has been come to be known as a *mathematical programming language*. Since Jack Schwartz' first observation of the relationship between SETL and some important topics in mathematics, there has developed, in the last 10 years: an educational variant of SETL, called ISETL, research papers in undergraduate mathematics education, textbooks, course development, NSF grants, a network of ISETL users, and a network of faculty implementing courses based on ISETL. ISETL has become a well known language in mathematics education circles in the US and abroad.

Using SETL to help student learn mathematics

I began in 1981 with SETL in a course on discrete mathematics. As time went on, there were developments, both in the language itself and in ideas on how to use it, generated by a growing number of students and faculty who were trying something different in mathematics education.

Learning mathematics by learning SETL syntax

From the beginning, it seemed that just learning SETL syntax helped students develop mathematical concepts. At a very elementary level students wrote code that performed operations on integers, floating point numbers and boolean values. This seemed to make them more comfortable with the general notion of operations on mathematical objects and new operations, for example modular arithmetic, came fairly easily.

A little more sophisticated, the SETL syntax for sets and tuples of numbers and the different properties of these objects helped students think about more than one number at a time and the different ways in which collections of numbers could be structured.

Learning the syntax for procedures in SETL (and later, funcs in ISETL) and using these programming structures to represent specific functions visibly enrichened students' repertoires of examples of functions. (See Breidenbach, Dubinsky, Hawks, and Nichols, 1992 for some details.) Moving over to (finite) sets of ordered pairs and learning the syntax for evaluating a function — which is identical to the syntax for functions defined as procedures — led to students beginning to develop understandings of functions that went beyond what most people see when teaching these topics.

The effect was striking and it led those of us who were observing it to wonder about the mechanism by which student learning appeared to be improving.

Making the abstract more concrete

Our first thoughts were that students had a lot of trouble with mathematical ideas that were abstract in the sense that they were not tied to what we can access through our senses. If you write down a set former such as the following,

$$\{x : x \in \{2, 3, \ldots, 100\} | (\nexists y \in \{2, 3, \ldots, x-1\} \ni x \bmod y = 0)\}$$

a mathematician will mentally convert the symbols to a structured collection of mathematical objects and relations to conclude that this expression calculates the set of primes less than 100. Beginning students will not find that so easy and will treat the expression as a somewhat mysterious collection of symbols. This seems to be the case even for students who know what the symbols represent individually and understand each relationship.

The trouble seems to be that this expression is abstract in the the sense of not being tied to concrete knowledge of the student. We observe that the situation is much improved if students are asked to represent the same set in ISETL syntax. It looks like this.

```
{x :  x  in  {2..100} |  (not exists y in {2,3..x-1} | x  .mod  y = 0)}
```

The fact that this syntax is so close to the mathematical notation not only makes the instructor happy, but also minimizes the programming overhead for students.

It seems that writing and running such expressions helps students understand the mathematics in that it becomes more concrete in the students' mind. That is, the student can imagine what is going on inside the computer as a statement such as this is processed and often, what is going on inside the computer can be described in ways that are close to what we would like to be going on inside the minds of our students.

But there is more to it than that and we have tried to increase our understanding of what goes on in people's minds when they learn mathematics or when they try but do not succeed in learning. We have been doing this, not just to satisfy our curiosity about why SETL (and ISETL) are so helpful for students trying to learn mathematics. We also believe that the more we learn about this mechanism, the more powerful kinds of instruction can be designed to facilitate it.

Mental constructions to learn mathematics

In parallel with the development of ways of using programming to help students learn mathematics, we have been working on a theory of learning which informs all of our pedagogical strategies. We will say a little more about this theory later in the paper but, basically, it has led us to study specific mental constructions that can be useful in learning mathematics. The application of this theory consists of designing instruction so as to foster students' making thse constructions. Often these constructions can be fostered by writing computer code.

Thus, if we wish students to think of functions as input/output processes, then writing a computer program to implement functions can have a salutory effect. Consider, for example, the classical calculus problem concerned with taking a rectangle of length l and width w and cutting out equal squares from each corner to make a box. The usual problem is to maximize the volume. However, for many students, the difficulty is to see, in this static situation, a function

which transforms the side s of the chosen square to a volume V. Pictures might help with this but our theory predicts that students' ability to think in terms of such a function is greatly enhanced if they are asked to write a computer program which accepts s and returns V. Such a program is very simple, of course:

```
V := func(s);
        return s*(l-2*s)*(w-2*s);
     end;
```

We find that students are helped even in such elementary cases, but the real payoff comes in situations such as piece-wise continuous functions and functions defined by integrals.

Often in mathematics, it is necessary to think about a function as an object which can be manipulated. This object interpretation can be very difficult for students and we will discuss in some detail below how a programming language can be used to help students develop the ability to interpret functions as objects. For now, consider the following example of a func in ISETL which accepts any computer representation of a function `f` and returns a function which approximates its derivative.

```
df := func(f);
         return func(x);
                  return (f(x+ 0.000001) - f(x))/0.000001;
                end;
      end;
```

We shall discuss further in the sequel how writing programs such as this one can help students develop the ability to to think of functions as objects.

ISETL vs. programming

All of the programming examples we give in this paper are in ISETL. As described below, our educational activities began with SETL and our work led to the development of ISETL. Within the world of mathematics education, some people think of those of us involved with these projects as "ISETL people". This is not strictly accurate. There are certain features a programming language must have in order to be considered a *mathematical programming language*. Our position is that any language which enjoys the following properties is, more or less, a *mathematical programming language*.

- The syntax is reasonably close to one or another standard mathematical notation system. It should be possible to write code so that items not necessary to the mathematics involved occur only rarely.
- Certain mathematical features are supported together with their usual mathematical properties. This includes finite sets and finite sequences with elements from any data types (including themselves), logical connectives and quantifiers, functions as procedures and as sets of ordered pairs, relations as sets of ordered pairs.
- All data types are first class objects in the sense that they can appear in any expression provided that it makes mathematical sense. In particular, sets and sequences can contain elements of different data types and procedures can be inputs and outputs to procedures.

ISETL has all of these features to a very high degree and I don't know of any other language that does so nearly as well. SETL of course comes closest, but in SETL, procedures are not first class objects. The computer algebra system MAPLE has a programming language component which, in recent versions, has acquired some of these features and a significant amount of what is done with ISETL can be done with MAPLE V. I have no doubt that this amount will increase with future enhancements of MAPLE.

The main point to be made is that the ideas expressed in this paper can be implemented with any mathematical programming language. It is only a temporary fact that ISETL is the closest to our ideal and hence for the time being, almost all of the examples are in ISETL. But it is our intention that the ideas involved should transcend any particular programming language. Our debt to Jack Schwartz is that he produced the first example — SETL — which supported educational work that has led and will lead to continued development of mathematical programming languages.

Overview of the movement

Here is a very rough chronology of the development of the movement to use ISETL in mathematics education.

1981-1985. First steps of using SETL informally in individual courses in discrete mathematics and abstract algebra at Clarkson University.

1985-86. Discussions with Gary Levin who developed ISETL version 1.0.

1986. First working version, ISETL 1.0 used in discrete math and abstract algebra courses at Berkeley and Clarkson.

1988 First ISETL-based textbook, *Learning Discrete Mathematics with ISETL*, written by Nancy Baxter, Ed Dubinsky and Gary Levin, published by Springer-Verlag.

1988. First NSF grant received for research and development of an ISETL-based course in Calculus. This project has received continuing and substantial support from NSF and is currently funded through 1996.

1990. Completion by Gary Levin of ISETL version 3.0 with interactive editing, formatted printing and the removal of many errors in earlier versions.

1991 Beginning three-year NSF funding for research and development of an ISETL-based course in Abstract Algebra.

1992 Publication of an ISETL-based textbook, *Calculus, Concepts, and Computers* by Ed Dubinsky and Keith Schwingendorf, published by West Educational Publishing.

1992. First summer workshop (17 days) for faculty to learn how to use ISETL-based approach to teaching calculus. These workshops have continued every summer since and are funded through 1996.

1992 Publication of an ISETL user's manual *ISETL: A Language for Learning Mathematics* by Jennie Dautermann, published by West Educational Publishing.

1993 Publication of an ISETL-based textbook, *Learning Abstract Algebra with ISETL*, written by Ed Dubinsky and Uri Leron, published by Springer-Verlag.

1994. First summer workshop (10 days) for faculty to learn how to use ISETL-based approach to teaching abstract algebra. This workshop is scheduled to run again in 1995.

Networks have been set up for ISETL users, implementers of the calculus course and implementers of the abstract algebra course. As of early 1995 it is estimated that more than 100 teachers in over 75 colleges (and a few high schools) throughout the world are using ISETL in one or more mathematics courses, involving a few thousand students.

It is fair to say that in this 15-year period, ISETL has become an established feature of undergraduate mathematics education. It has not become what one might call "widespread", but it is known and respected, especially among those who are engaged in postsecondary curriculum reform in mathematics.

SETL and ISETL

When people first worked with SETL, the emphasis was on expressive power and functionality. The most important goal was that very complex programs could be written relatively quickly and easily, essentially by stating the required relationships in mathematical language. It was suggested that the paradigm under which SETL worked was that the "statement of the problem is the program". There was less concern with speed and one heard it said about a complex SETL program that, "If it moves, it's fast enough." Later, there was some concern with efficiency and the Data Representational Sublanguage (Dubinsky, Freudenberger, Schonberg, and Schwartz, 1989) was designed to help make programs written in SETL run faster and more efficiently.

The initial reactions of students who used SETL in the first courses was that it helped considerably. But there were properties of SETL that made it clear that its use in education for beginning mathematics courses would be highly limited. Indeed, it was the early indications of the educational potential of such a language that provided the impetus for designing a variation of SETL focussed on educational uses.

The main problematic features of SETL in 1985 were:

- SETL is very large and is not useable on the small personal computers just beginning to become popular in academia.
- SETL is slow.
- SETL is a compiled language and it was necessary to operate in batch mode.
- SETL has many complex features that are not essential for educational purposes.
- Functions are not first class objects in SETL.

In summer 1985, I began conversations with Gary Levin about the possibility of developing an interactive version of SETL that would eliminate these problems and be more appropriate for educational uses. Discussions continued throughout the Fall and by January, 1986 we had agreed on specifications and Gary began to work on what would become (with the agreement of Jack Schwartz) ISETL. By April, 1986 Levin had produced a version which ran but with

a multitude of errors and just before the semester ended it had proceeded far enough that I could show it to my discrete mathematics class where the students had been using SETL. By Fall of 1988, ISETL version 1.0 was sufficiently stable that I could use it in courses on discrete mathematics and abstract algebra.

Following are the main features of ISETL that distinguish it from SETL.

- Although most of the syntax is identical to SETL, some features of SETL are not implemented in ISETL.
- ISETL is interactive and response is quick.
- ISETL runs on PCs under MS-DOS, Macintosh and SUN Workstations.
- ISETL is under 250K so that it does not require very large computers.
- In ISETL, functions are first class objects.

It is worth noting that although Gary Levin retains ownership of ISETL, it is available free to anyone who requests it. A disk is distributed with the several textbooks and manuals that have been published. Copies can be obtained, for example, from the author.

2 Learning Theory and ISETL

Before discussing our theory of learning we must first describe a paradigm for combining this theory with empirical investigations and development of instructional approaches. Then we will discuss the theory in somewhat general terms. Finally, in describing the nature of mental constructions involved we will try to indicate how the theory fits with learning mathemtics.

A paradigm for research and curriculum developent

The paradigm which is used in our investigations is a repeated traversal of a circle of activities that can be illustrated as in Figure 1.

Figure 1: Paradigm for research and curriculum development

Investigation of a particular topic begins with a theoretical analysis based on a general theory of learning (which I will discuss in the next paragraph), the researchers' own knowledge of the mathematics involved, and any informal observations of students, for example, in teaching

the material in a traditional way. The purpose of this analysis is to propose, in a preliminary and tentative manner, a genetic decomposition of the concept in question. That amounts to suggesting specific mental constructions which a student can make in order to learn the concept. The next step is to design and implement instruction aimed at getting students to make the proposed mental constructions. As the students are experiencing this instruction, data is collected in several different ways, using both quantitative and qualitative methods of gathering information.

The final step in the traversal is to coordinate the empirical data obtained with the theoretical analysis. This means, on one hand, that the theoretical analysis suggests questions to ask of the data: to wit, does it appear that the proposed mental constructions were made by students? Focusing the analysis of data in this way can help the researcher deal with a huge amount of information meaningfully, but far from exhaustively. Indeed we have very often used the same data more than once in studies differentiated by the questions to which a theoretical analysis points us.

On the other hand, it sometimes occurs that the mental constructions students appear to be making are different from what has been proposed. One possible reaction to this is to reconsider one or more aspects of the theoretical analysis.

To summarize, the theoretical analysis drives the instruction which creates the data. The theoretical analysis directs the analysis of data and is simultaneously subject to revision as a result of this data analysis. This circle of activity is then repeated with the (possibly new) theoretical analysis. It is repeated as often as appears necessary to understand the epistemology of the particular topic.

If things work properly, then learning should improve in a natural way as result of instruction that relates to how the students can learn the concept or concepts.

In the remaining paragraphs of this section I will sketch the general theory in its present stage of development and describe the nature of mental constructions that the theory proposes along with some examples.

Mathematical knowledge and its acquisition

Our theory begins with a statement of what it means to learn and know something in mathematics.

> An individual's mathematical knowledge is her or his tendency to respond to mathematical problem situations by reflecting on them in a social context and constructing or reconstructing mathematical actions, processes and objects and organizing these in schemas to use in dealing with the situations.

There is, in this statement, references to a number of aspects of learning and knowing. For one thing, the statement acknowleges that what a person knows and is capable of doing is not necessarily available to her or him at a given moment and in a given situation. All of us who have taught (or studied) are familiar with the phenomenon of a student missing a question completely on an exam and then really knowing the answer right after, without looking it up. A related phenomenon is to be unable to deal with a mathematical situation but, after the slightest suggestion from a colleague or teacher, it all comes running back to your consciousness.

Thus, in the problem of knowing, there are two issues: learning a concept and accessing it when needed.

Reflection is an important part of both learning and knowing. Mathematics in particular is full of techniques and algorithms to use in dealing with situations. Many people can learn these quite well and use them to do things in mathematics. But understanding mathematics goes beyond the ability to perform calculations, no matter how sophisticated. It is necessary to be aware of how these procedures go, to get a feel for the result without actually performing all the calculations, to be able to work with variations of a single algorithm and to understand relationships among algorithms.

It is a controversial point, but this theory takes the position that reflection is best performed in a social context. There is evidence in the literature (Vidakovic, 1993) for the value to students of social interaction and there is also the cultural fact that almost all research mathematicians feel very strongly the need for interactions with colleagues before, during, and after creative work in mathematics.

This theory asserts that "possessing" knowledge consists in a tendency to make mental constructions that are used in dealing with a problem situation. Often the construction amounts to reconstructing (or remembering) something previously built so as to repeat a previous method. But progress in the development of mathematical knowledge comes from making a reconstruction in a situation similar to, but different in important ways from, a problem previously dealt with. Then the reconstruction is not exactly the same as what existed previously, and may in fact contain one or more advances to a more sophisticated level. This whole notion is related to the well known Piagetian dichotomy of assimilation and accomodation. (Piaget, 1992)

Finally, the question arises of what is constructed, or what is the nature of the constructions and the ways in which they are made? It is when we talk about this that our theoretical perspective, which may appear applicable to any subject whatsoever, becomes specific to mathematics. We will deal with this question in the next paragraph.

Mental constructions for learning mathematics

As indicated in Figure 2, understanding a mathematical concept begins with manipulating previously constructed mental or physical objects to form actions; actions are then interiorized to form processes which are then encapsulated to form objects. Objects can be de-encapsulated back to the processes from which they were formed. Finally, processes and objects can be organized in schemas.

Action. A transformation is considered to be an action when it is a reaction to stimuli which the subject perceives as external. This means that the individual requires complete and understandable instructions giving precise details on steps to take in connection with the concept.

For example, a student who is unable to interpret a situation as a function unless he or she is *given* a (single) formula for computing values is restricted to an action concept of function. In such a case, the student is unable to do very much with this function except to evaluate it at specific points and to manipulate the formula. For example, functions with split domains, inverses of functions, composition of functions, sets of functions and

Figure 2: Constructions for mathematical knowledge

the notion that the derivative of a funtion is a function, or the solution of a differential equation is a function are all sources of great difficulty for students.

Another example of an action conception comes from the notion of a (left or right) coset of a group in abstract algebra. Consider, for example, the modular group $[\mathcal{Z}_{20}, +_{20}]$ — that is, the integers $\{0, 1, 2 \ldots, 19\}$ with the operation of addition mod 20 — and the subgroup $H = \{0, 4, 8, 12, 16\}$ of multiples of 4. It is not very difficult for students to work with a coset such as $2+H = \{2, 6, 10, 14, 18\}$ because it is formed either by a listing of the elements according to some rule ("begin with 2 and add 4") or an explicit condition such as, "the remainder on division by 4 is 2". This is an action conception. Something more is required to work with cosets in a group such as $\mathcal{S}_n$, the group of all permutations on n objects where simple formulas are not available. Even in the more elementary situation of $\mathcal{Z}_n$ students will have difficulty in reasoning about cosets (such as counting them, comparing them, etc.)

According to this theory, all of these difficulties are related to students' inability to interiorize these actions to processes, or encapsulate the processes to objects.

Although an action conception is very limited, it is an important part of the beginning of understanding a concept. Therefore, instruction should begin with activities designed to help students construct actions.

Process. When an individual reflects on an action scheme and interiorizes it then the action can become perceived as a part of the individual and he or she can establish control over it.

In the case of functions, a process conception allows the subject to think of a function as receiving one or more inputs, or values of independent variables, performing one or more operations on the inputs and returning the results as outputs or values of dependent variables. In this conception it is not necessary to explicitly perform the operations; it is enough to imagine them in greater or less detail.

Thus, with a process conception of function, an individual can compose two or more processes to construct the composition, or reverse the process to obtain inverse functions.

Object. When an individual reflects on operations applied to a particular process, becomes aware of the process as a totality, realizes that transformations (whether they be actions

or processes) can act on it, and is able to actually construct such transformations, then he or she is thinking of this process as an object.

In the course of performing an action or process on an object, it is often necessary to de-encapsulate the object back to the process from which it came in order to use its properties in manipulating it.

For example, given an element x and a subgroup H of a group G, if an individual thinks generally of the (left) coset of x modulo H as a process of operating with x on each element of H, then this process can be encapsulated to an object xH. Then actions on cosets of H, such as counting their number, comparing their cardinality, and checking their intersections can make sense to the individual. Thinking about the problem of investigating such properties involves the interpretation of cosets as objects whereas the actual finding out requires that these objects be de-encapsulated in the invidual's mind so as to make use of the properties of the processes from which these objects came (certain kinds of set formation in this case.)

It is easy to see how encapsulation of processes and de-encapsulating them back to objects arises when one is thinking about manipulations of functions such as adding, multiplying, or just forming sets of functions.

In general, encapsulating processes to become objects is considered to be extremely difficult (Sfard, 1987) and not very many pedagogical strategies have been effective in helping students do this in situations such as functions or cosets. A part of the reason is that there is very little in our experience that corresponds to performing actions on what are interpreted as processes.

Schema. Once constructed, objects and proccesses can be interconnected in various ways: for example, two or more processes may be coordinated by linking them through composition or in other ways; processes and objects are related by virtue of the fact that the former acts on the latter. A collection of processes and objects can be organized in a structured manner to form a schema. Shemas themselves can be treated as objects and included in the organization of "higher level" schemas.

For example, sets and binary operations are linked to form pairs which may or may not satisfy certain properties. All of this can be organized to construct the schema for group. Groups and rings and other such mathematical objects might be organized in a schema called algebraic structures.

3 Examples of ISETL in mathematics education

We will organize our examples around the three kinds of mental constructions described in the previous section: actions, processes and objects. At the same time, we will try to indicate, in these examples, some of the features of full instructional treatments of various topics in undergraduate mathematics. These will include mathematical induction, predicate calculus, the fundemental theorem of calculus, sequences and series, binary operations, cosets and quotient groups.

At the end of this section we will give a very brief description of how the work with ISETL is integrated into a standard undergraduate mathematics course. We do this through a pedagogical structure called the ACE Teaching Cycle.

Programming to make mental constructions

Note that actions require objects, objects require processes, and processes require actions so there is a certain circularity or spiral aspect to these considerations.

Actions

Following is a set of ISETL instructions as they would appear on the screen followed by the computer's response. It is taken from the first pages of the textbook used in the calculus project. (Dubinsky, Schwingendorf, and Mathews, 1995.) The > symbol is the ISETL prompt and lines which begin with this symbol or >>, which indicates incomplete input, are entered by the user. Lines without these prompts are what the computer prints on the screen.

```
> 7+18;
25;

> 13*(233.8);
-3.03940e+003;

> 5 = 2.0 + 3;
true;

> 4 >= 2 + 3;
false;
> 17
> + 23.7 - 46
> *2;
> ;
-5.1300e+001;
> x := -23/27;
> x;
-6.21622e-001;

> 27/36;
7.50000e-001;

> p := [3,-2]; q := [1,4.5]; r := [0.5,-2,-3];
> p; q; r;
[3, -2];
[1, 4.50000e+000];
[5.00000e-001, -2, -3];

> p(1); p(2); q(2); r(3); 3;
-2;
4.50000e+000;
-3;

p(1)*q(1) + p(2)*q(2);
-6.00000e+000;

> length := 0;
> for i in [1..3] do
>   length := length + r(i)**2;
>   end;
> length := sqrt(length);
> length;
3.64005e+000;
```

Students are asked to do these exercises for the purpose of becoming familiar with the syntax of ISETL, but at the same time, there are several mathematical concepts which they have an opportunity to construct at the action level. For example, there are simple propositions, the

formation of pairs and triples of numbers and the action of picking out an indexed term of a given sequence. Also the concept of dot product appears as an action. Finally, the algorithm for computing the length of a vector in three dimensions appears as an action because the calculation is explicit and is applied to a single vector.

Following are some examples from the first pages our textbook for an Abstract Algebra course (Dubinsky and Leron, 1994.) Students are asked to guess what will happen when this code is run, run it, and then explain discrepancies between their predictions and the results.

```
>    A := "Abstract Algebra";
>    A(1);
"A";
>    A(4); A(9);
"t";
" ";
>    A(11); A(6); A(10);
"l";
"a";
"A";
>    is_string(A);   is_string(A(6));
true;
true;

>    A = A(10);
false;
>    B := "ABSTRACT"; C := "AB" + "STRACT";
>    B = C;
true;

>    "B" in "ABS"; "b" in "ABS";
true;
false;

>    (2/=3) and ((5.2/3.1) > 0.9);
>    (3 <= 3) impl (3 = 2 +1);
>    (3 <= 3) impl (not (3 = 2 +1));
>    (3 > 3) impl (3 = 2 +1);
>    (3 > 3) impl (not (3 = 2 +1));

>    7 mod 4; 11 mod 4; -1 mod 4;
>    (23 + 17) mod 3;

>    x := 4; y := 2;
>    if (x + y) mod 6 = 0 then
>        ans := "Additive Inverses!";
>    end;
>    ans;
```

Here we have simple string operations which the students can understand as actions, elementary operations on propositions, tests and choices among alternatives based on modular

arithmetic, and more string operations. The fact that everything is very explicit and tied to single examples makes it likely that students will construct their understandings of these various activities as actions. Although actions are the most primitive kind of mental constructions, the computer allows students to construct them in more advanced mathematical situations than is possible with pencil and paper.

Processes

Consider now the following list of pairs `G`, `op` of sets and operations.

1. `G` is `Z12` (the integers mod 12) and `op` is `a12` (addition mod 12).
2. `G` is `Z12` (the integers mod 12) and `op` is `m12` (multiplication mod 12).
3. `G` is `2Z12` (the even integers mod 12) and `op` is `m12`.
4. `G` is `Z12 - {0}` and `op` is `m12`.
5. `G` is `Z5` (the integers mod 5) and `op` is `m5` (multiplication mod 5).
6. `G` is `Z5 - {0}` and `op` is `m5`.
7. `G` is `S3` (the set of permutations of $\{1, 2, 3\}$) and `op` is composition of permutations.

Initially, students are asked merely to express them in ISETL. Writing such code helps the students to construct the idea of a binary operation as an action, although, in order to make sense of the problem, each individual set and operation must be seen as a process which has been encapsulated as an object.

In our abstract algebra course the student meets this list just after having written some short programs to check various operations such as closure. Applying these programs to the individual pairs and thinking about what ISETL does, for example, in running the code

```
is_closed(S12, A12);
```

where the following ISETL definitions have been made previously,

```
Z12 := 0..11;
a12 := | x,y -> (x+y) mod 12|;

is_closed := func(S,op);
    return forall x,y in S | x .op y in S;
end;
```

and thinking about what the computer is doing when running this code, helps the student move from action to process.

The student writes little programs like this for each of the standard properties of binary operations and applies them to long lists of examples. In this way, each of these properties is constructed by the student as a process in her or his mind. It is a process and not just an action because it is not specific code applicable to just one example, but rather a computer function (called `func` in ISETL) that can take an arbitrary set and binary operation as input and return the truth value of the assertion that the binary operation possesses the property.

This can be done for more advanced ideas as well. For example, the student might write the following code in dealing with cosets of a subgroup of S_4 the group of permutations of four objects.

```
S4 := {[a,b,c,d] :  a,b,c,d in [1..4] | #{a,b,c,d} = 4};

op := func(p,q); return [p(q(i)) :  i in [1..4]];

H := {[1,2,3,4], [2,1,4,3, [3,4,1,2], [4,3,2,1]};

x := [2,1,3,4];
xH := {x .op y | y in H};
xH;
Hx := {y .op x | y in H};
Hx;
xH = Hx;
```

The result of running this code is to display the left and right cosets of x mod H and the truth value of the proposition that these two sets are equal. Again, writing and running this code helps students understand cosets as processes of multiplying a single element by every element of a given set, because the computer work has gotten them to make formation of a coset a mental process that exists in their minds.

The method is applicable a wide variety of mathematical situations. For example, in calculus, students overcome their resistance to piece-wise defined functions when they write programs in which the definition is implemented through the simple use of a conditional.

```
f := func(x);
        if x <= 1 then return 2-x**2;
        else return x/2 + 1.5;
        end;
    end;
```

Working with pointwise sums, products and even compositions of such functions helps students construct a process conception of function. Research suggests that this can make a difference (Breidenbach et al, 1992.)

Finally we might mention the example of proposition-valued function of the positive integers. Our research suggests that one of the difficulties students have with proof by induction is at the very beginning. A student is faced with a problem: show that a certain statement involving an arbitrary integer is true for all (sufficiently large) values of the integer. This kind of problem is very new and different for most students. It really is a (mental) function which accepts a positive integer and plugs it into the statement to obtain a proposition which may be true or

false — and the answer could be different for different values of n. Once again, expressing this problem as an ISETL `func` is a big help for students in figuring out how to begin.

Suppose, for example, that the problem is to determine if a gambling casino with only \$300 and \$500 chips can honor any amount of money within the nearest \$100. We encourage students to begin their investigation by writing a computer program that accepts a positive integer and returns a boolean value. Following is one solution they generally come up with in our elementary discrete mathematics course.

```
P := func(n);
        if is_integer(n)
                and n > 0
                and exists x,y in [0..n/3] | 3*x + 5*y = n;
        then return true;
        else reurn false;
        end;
     end;
```

Objects

Objects are obtained by encapsulation of processes and an individual is likely to do this when he or she reflects on a situation in which it is necessary to apply an action to a dynamic process. This presents a difficulty because the action cannot be applied to the process until *after* the process has been encapsulated to an object. In fact, mental constructions do not seem to occur in simple logical sequences and so it can happen that the need to create an object (in order to apply an action to a process), the encapsulation of a process to form the object, and the application of an action to that object (which was the source of the need) all happen together, initially in some amorphous combination with a gradual differentiation, reorganization and integration culminating in the clear application of the action to the object.

Getting students to do all of this is another matter and there are very few effective pedagogical methods here. As we have indicated, one such method is to put students in situations where a problem is solved or a task is performed by writing programs in which the processes to be encapsulated are inputs and/or outputs to the programs.

For example, forming a pair whose components are a set and a binary operation requires that these components be interpreted as objects. When the student goes on to write a program (such as `is_closed`) which accepts such a pair, checks a certain property for that pair and returns `true` or `false`, then the pair is being treated as an object to which the process connected with the property is applied. This leads to the mental act of encapsulating the binary pair into an object. In the course of thinking about the process connected with the property (e.g., in writing the program), this object is de-encapsulated back into a set and a binary operation, each of which must in turn be de-encapsulated back to the processes from which they came.

A more complicated situation concerns the status of cosets, first as processes (discussed above) and then as objects. This is important for many situations in the student's first studies of group theory. Counting cosets, comparing their cardinalities, thinking about their intersections are essential components of Lagrange's theorem and all of these mathematical activities require that cosets be interpereted as objects. This is difficult for students and even more difficult is the idea of defining a binary operation on cosets.

Our pedagogical strategy for helping students overcome the difficulty of encapsulating cosets into objects is to ask them to write a program that accepts a binary operation pair `[G,o]` and returns a function `oo` of two variables. These two variables can have values which are any of the four combinations in which each variable is either an element of `G` or a subset of `G`. If both are elements, then `x .oo y` is the ordinary group operation. If one is a subset and the other is an element then `x .oo y` is the product of the element with all elements of the subset. If both are subsets then `x .oo y` is the set of all products of two group elements, the first taken from `x` and the second from `y`.

The solution to this problem is something like,

```
PR := funcG,o);
         return func(x,y);
             if x in G and y in G then return x .o y;
             elseif x in G and y subset G then return {x .o b:b in y};
             elseif x subset G and y in G then return {a .o y:a in x};
             elseif x subset G and y subset G then return {a .o b:a in x, b in y};
             end;
         end;
      end;
oo := PR{S4, o};
```

where the last line applies `PR` to the group of permutations of four objects.

Students find this problem very difficult although the reader will notice that the desired program is not very long or subtle. As far as we can tell, from analyzing the problem and talking to students, their difficulty seems to be that the whole thing makes no sense to them. We think that is because they are not understanding that subsets of `G` can be objects. They struggle with the problem, usually for about a week (while doing other things) in our abstract algebra course and most of them eventually succeed with more or less help from instructors. When it is over, we have found that understanding Lagrange's theorem and quotient groups becomes a reasonable expectation for students.

Two extremely important examples of construction of objects occur in Calculus in connection with derivatives and integrals. Our research suggests that, although it is very simple for mathematicians, the idea that the derivative of a function is a function is not immediate for students. Writing a program like the following, which accepts a function and returns an approximation to its derivative appears to help.

```
          df := func(f);
                   return func(x);
                            return (f(x + 0.00001) - f(x))/0.00001;
                   end;
                end;
```

Integration is more difficult. The idea of defining a function by using the definite integral with one limit of integration fixed and the other allowed to vary is a major stumbling block for calculus students. In our treatment of integration, students have written a program called `Riem` which accepts a function and a pair of numbers and computes an approxiation to the integral of the function over the interval determined by the points. The students are then asked to write the following program.

```
Int := func(f,a,b);
          return func(x);
                    if a <= x and x <= b then return Riem(f,a,x);
          end;
       end;
```

Using this program, students are able to construct and study approximations to the logarithm function and inverse trigonometric functions.

In our treatment of mathematical induction, students learn to treat propositions as objects and at the same develop an understanding of the "implication from n to $n+1$" as the object whose truth value as n varies is to be considered. The main tool that we use is to have them write and apply the following program which accepts a function whose domain is the positive integers and whose range is the two element set {`true,false`}. This program returns the corresponding implication valued function. (The symbol $ refers to a comment and anything after this symbol on the line in which it appears is ignored by ISETL.

```
implfn := func(P); $ P is a proposition-valued function.
             return func(n);
                       return P(n) impl P(n+1);
             end;
          end;
```

Schemas

Our use of ISETL to help students form schemas to organize collections of processes, objects, and other schemas is at the time of this writing, somewhat ad hoc. Roughly speaking, we ask students to write a set of computer programs that implements a mathematical concept and then to apply their code to specific situations.

For example, over a period of time, students will write the following programs to check that a subset is a subgroup and that it is normal. Then they write a program to construct the quotient group of a group mod a normal subgroup. Their programs might look like the following.

```
subgrp := func(G, op1, H, op2);
   return group(G,op1) and group(H,op2)
           and (H subset G) and
           forall x1,y1 in G, x2,y2 in H |
            (x1 .op y1) = (x2 .op2 y2);
   end;



normal := func(G, op1, H, op2);
   oo := PR(op1,G);
   return subgrp(G, op1, H, op2) and
           forall g in G | g .oo H = H .oo g;
   end;



quotient := func(G, op1, H, op2);
   oo := PR(op1,G);
   if normal (G, op1, H, op2) then
           return [{g .oo H : g in G}, oo];
           end;
   end;
```

Then, the following code will construct the quotient of a particular normal subgroup of S_4. (Recall from above the definitions of `S4, op, H`.)

```
S4 := {[a,b,c,d] :  a,b,c,d in [1..4] | #{a,b,c,d} = 4};

op := func(p,q); return [p(q(i)) :  i in [1..4]];

H := {[1,2,3,4], [2,1,4,3, [3,4,1,2], [4,3,2,1]};


subgrp(S4,op,H,op);

[3,2,4,1] .Pr(op) H;

[3,2,4,1] .Pr(op) H = H .Pr(op) [3,2,4,1];


normal(S4, op, H, op);

S4modH := quotient(S4, op, H, op);

S4modH;
```

The schema for the Fundemental Theorem of calculus requires much less code but it is considerably more complicated. Having written the two funcs `df` to approximate the derivative and `Int` to approximate the integral (see above), students are asked to write code that will first do one and then the other, in both orders. This problem gives the students considerable difficulty

and they struggle with it for a long time. We feel that this is a useful struggle because it has to do with their ability to interpret functions as objects, to develop processes corresponding to differentiation and integration, and to put it all together in what is essentially a statement of the Fundemental Theorem.

The actual code to solve this problem is very short:

```
df(Int(f(a,b));

Int(df(f),a,b);
```

We ask students to apply their code to a specifc function and to construct a table with four columns: values of the independent variable, corresponding values of `f`, and corresponding values of the above two lines of code. When the example is a function that does not vanish at a, then the second and third columns are identical, but the fourth is different. The students see the point right away — all three columns are supposed to be the same, but they feel they have made an error in connection with the last column. After some investigation, they tend to discover on their own the idea of the "constant of integration".

Sometimes we don't ask students to write code but rather to investigate code which we provide. We do this in situations where the particular code involves more in the way of programming issues than mathematical issues This is the case for code which simulates the operation of induction. We give them the following code which makes use of the `func implfn` which they have written (see above) and is applied to a proposition valued function `P`. The first few lines find a starting point and the rest of the code runs through the induction steps. If the proposition does hold from the selected starting point on, then the code will run forever.

```
start := 1;
while P(start) = false do
    start := start + 1;
end;

L := [];
L(start) := true;
while L(n) = true and implfn(P)(n) = true do
    L(n+1) := true;
    n := n+1;
    print "The proposition P is true for n = ", n;
end; print "P is not proven for n = ", n+1;
```

The ACE Teaching Cycle

We have found that our way of having students work with computers requires a different structure for the course. We have developed a standard structure called the *ACE Teaching Cycle*. In this design, the course is broken up into sections, each of which runs for one week. During the week, the class meets on some days in the computer lab and on other days in a regular classroom in which there are no computers. Homework is completed outside of class. Usually, we have the students working in cooperative groups in all of these activities.

Following is a description of the three components of this structure with some indications of the pedagogical goals of each component.

Activities:

Class meets in a computer lab where students work in teams on computer tasks designed to foster specific mental constructions suggested by research. The lab assignments are generally too long to finish during the scheduled lab and students are expected to come to the lab when it is open or work on their personal computers, or use other labs to complete the assignment.

Class:

Class meets in a classroom where students again work in teams to perform paper and pencil tasks based on the computer activities in the lab. The instructor leads inter-group discussion designed to give students an opportunity to reflect on the calculations they have been working on. On occasion, the instructor will provide definitions, explanations and overviews to tie together what the students have been thinking about.

Exercises:

Relatively traditional exercises are assigned for students to work on in teams. These are expected to be completed outside of class and lab and they represent homework that is in addition to the lab assignments. The purpose of the exercises is for students to reinforce the ideas they have constructed, to use the mathematics they they have learned and, on occasion, to begin thinking about situations that will be studied later.

4 Conclusion

We have tried to show how Jack Schwartz's original idea for, and development of, a programming language based on mathematics, that is easy to program in, and is useful for solving major problems in computer science, led to a variation that provides an educational tool of critical value in one approach to designing instruction that relates to our understanding of how people can learn mathematical concepts. The variety of mathematical situations in which this approach can be used is indicated in the examples we have given from abstract algebra, calculus and mathematical induction. Results of using this pedagogical strategy can be found in the literature. The data suggests that this can be a very powerful way of helping students learn mathematical concepts.

There are continuing efforts in the basic research into learning that is discussed in this paper and the development of corresponding pedagogy using ISETL. The use of ISETL is firmly established in mathematics education and we can hope that at the next festchrifft in honor of Jack Schwartz we can report on widespread acceptance of this method.

REFERENCES

D. Breidenbach, E. Dubinsky, J. Hawks, and D. Nichols, *Development of the process conception of function*, Educational Studies in Mathematics, 23 (1992), 247-285.

E. Dubinsky, S. Freudenberger, E. Schonberg, and J.T. Schwartz, *Reusability of design for large software systems: An experiment with the ISETL optimizer*, in **Software Reusability I**, (T. Biggerstaff and A. Perlis, eds.), New York:ACM Press, Addison-Wesley, 1989, pp. 275-293.

E. Dubinsky and U. Leron, **Learning Abstract Algebra with ISETL**, New York:Springer-Verlag, 1994.

E. Dubinsky, K. E. Schwingendorf, and D. M. Mathews, **Calculus, Concepts and Computers**, 2nd edition, New York:McGraw-Hill, 1995.

J. Piaget, **The principles of Genetic Epistemology** (W. Mays trans.) London: Routledge& Kegan Paul, 1972. (Original published 1970).

A. Sfard, *Two conceptions of mathematical notions, operational and structural*, Proceedings of the 11^{th} Annual Conference of the International Group for the Psychology of Mathematics Education, (A. Borbàs, ed.) Montreal, (1987) 162-169.

D. Vidakovic, *Differences between group and individual processes of construction of the concept of inverse function*, unpublished doctoral dissertation, Purdue University, 1993.